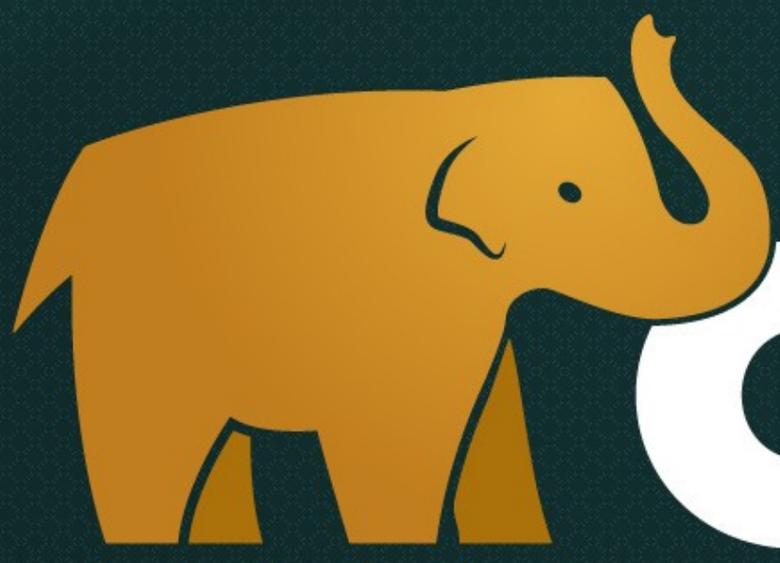


ceylon

Introduction to Ceylon

Stéphane Épardaud

Emmanuel Bernard

Red Hat

Executive summary

- What is Ceylon
- Why Ceylon
- Features and feel
- Demo
- The community
- Status

About Emmanuel Bernard

- Hibernate
- JCP
- Podcasts
  - JBoss Community Asylum (http://asylum.jboss.org)
  - Les Cast Codeurs (http://lescastcodeurs.com)
- The rest is at http://emmanuelbernard.com
- @emmanuelbernard

About Stéphane Épardaud

- Open-Source projects
  - RESTEasy, Ceylon
  - jax-doclets, Play! modules, Stamps.js
- Ceylon contributor since…
  - 13 May 2011 (one month after Ceylon hit SlashDot)
  - compiler, ceylondoc, Herd

- http://stephane.epardaud.fr

Origins of Ceylon

- Initiated and lead by Gavin King
- Improve upon frustrations of Java
- Help by others at JBoss
  - Max, Emmanuel, Pete etc
- Starting blocks
  - on the JVM
  - in the spirit of Java
  - practical

What is Ceylon?

- Ceylon is
  - Powerful, readable, predictible
- Ceylon has
  - A platform, modularity, tooling

Introduction to Ceylon

A boring class

- Looks familiar, right?

```
class Rectangle() {
    Integer width = 0;
    Integer height = 0;

    Integer area(){
        return width * height;
    }
}
```

A Real Ceylon class

- No (big) surprise

```
shared class Rectangle(width, height){

    shared Integer width;
    shared Integer height;

    shared Integer area(){
        return width * height;
    }
}
```

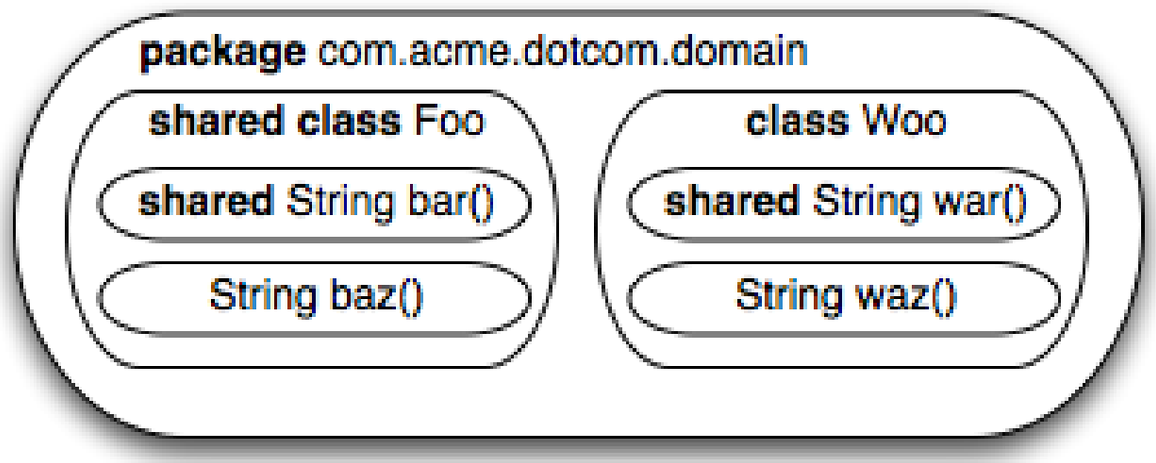
Where is my constructor?

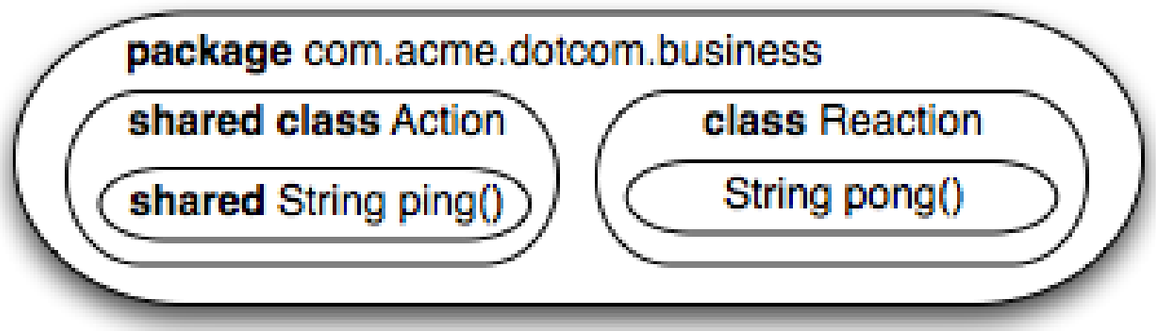
- In the class body

```
shared class Rectangle(width, height) {
    shared Integer width;
    shared Integer height;

    // it is here!
    if(width == 0 || height == 0){
        throw;
    }

    shared Integer area(){
        return width * height;
    }
}
```

First differences

- Simpler and more regular access rules
  - No `protected`, `package`, `private`
  - `shared` = public-ish, otherwise scope-private





Attributes

- Immutable by default

```
class Circle(){
    Integer scale = 1;
    variable Integer radius := 2;
    radius++;
    Integer diameter {
        return radius * 2;
    }
    assign diameter {
        radius := diameter / 2;
    }
}
```

Attributes

- Unless marked variable
- Assigned with :=

```
class Circle(){
    Integer scale = 1;
    variable Integer radius := 2;
    radius++;
    Integer diameter {
        return radius * 2;
    }
    assign diameter {
        radius := diameter / 2;
    }
}
```

Attributes

- Getter/setter without carpal tunnel syndrome

```
class Circle(){
    Integer scale = 1;
    variable Integer radius := 2;
    radius++;
    Integer diameter {
        return radius * 2;
    }
    assign diameter {
        radius := diameter / 2;
    }
}
```

Inheritance

```
shared class Point(x, y) {
    shared Integer x;
    shared Integer y;
}

shared class Point3D(Integer x,
                     Integer y, z)
        extends Point(x, y) {
    shared Integer z;
}
```

Abstractions

- Method, attributes and classes can be overridden
  - Factory pattern
- Can't override by default
  - `default`: can be overridden, has a default impl
  - `formal`: must be overridden, with no default impl
- `@Override` in Java => `actual` in Ceylon
  - Non optional

Abstractions (example)

```
abstract class Shape(){
    shared formal Integer area();
    // magic: this is toString()
    shared actual default String string {
        return "Abstract area: " area.string " m2";
    }
}
class Square(Integer width) extends Shape(){
    shared actual Integer area() {
        return width * width;
    }
    shared actual String string
      = "Square area: " area.string " m2";
}
```

Overloading

- No Overloading
  - WTF!?
- Overloading is evil

You need overloading...

- To support optional parameters
  - Ceylon has them
  - Even named-parameters
- To work on different (sub)types of parameters
  - Not safe if a new type is introduced
  - Ceylon has union types and type cases

Optional and named parameters

```
class Rectangle(Integer width = 2,
                Integer height = width * 3){
    shared Integer area(){
        return width * height;
    }
}
void makeRectangle(){
    Rectangle rectangle = Rectangle();
    Rectangle rectangle2 = Rectangle {
        height = 4;
        width = 3;
    };
}
```

Type based switch case

```
void workWithRectangle(Rectangle rect){}
void workWithCircle(Circle circle){}
void workWithShape(Shape shape){}

void supportsSubTyping(Shape fig){
    switch(fig)
    case(is Rectangle){
        workWithRectangle(fig);
    }
    case(is Circle){
        workWithCircle(fig);
    }
    else{
        workWithShape(fig);
    }
}
```

Keeping it DRY

```
interface Figure3D {
    shared formal Float area;
    shared formal Float depth;
    shared formal Float volume;
}
```

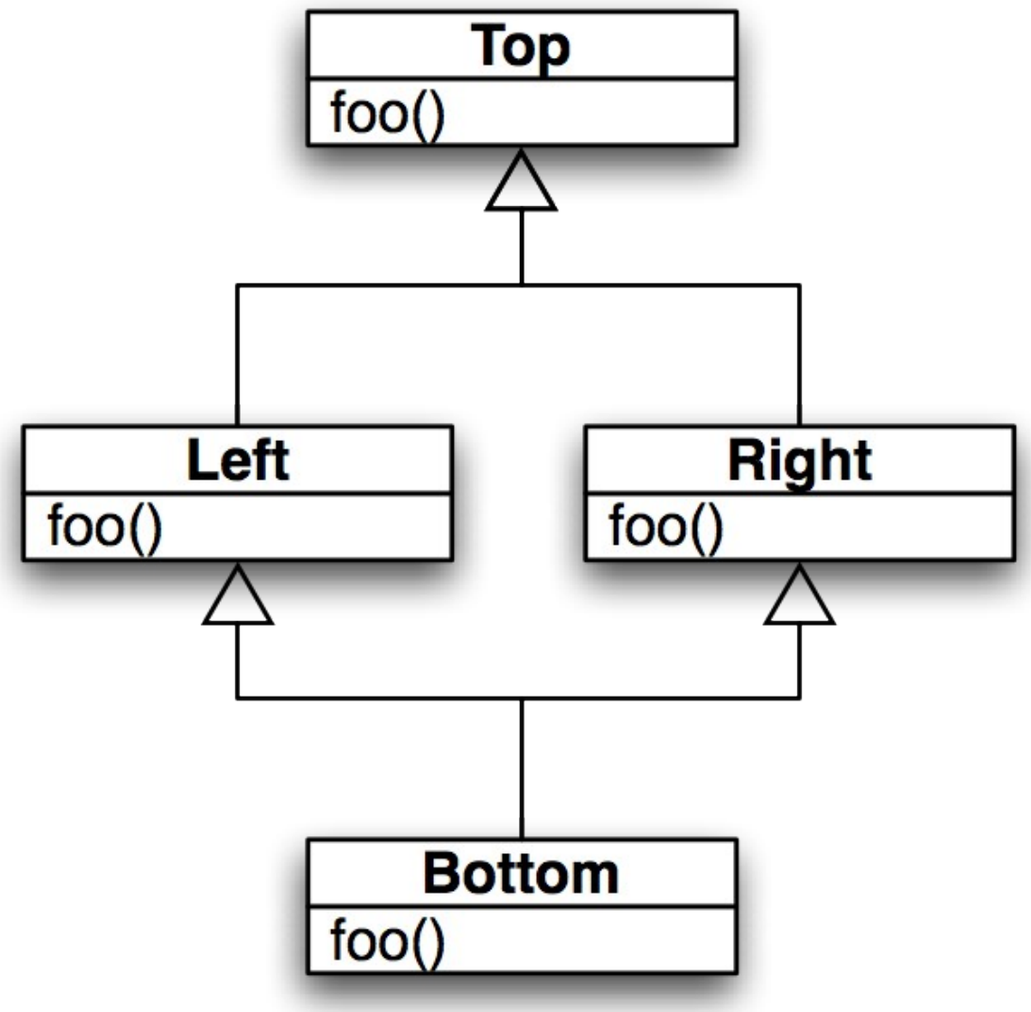
```
class Cube(Float width) satisfies Figure3D {
    shared actual Float area = width * width;
    shared actual Float depth = width;
    shared actual Float volume = area * depth;
}

class Cylinder(Integer radius, depth)
        satisfies Figure3D {
    shared actual Float area = 3.14 * radius ** 2;
    shared actual Float depth = depth;
    shared actual Float volume = area * depth;
}
```

Interfaces with implementation

```
interface Figure3D {
    shared formal Float area;
    shared formal Float depth;
    shared Float volume {
        return area * depth;
    }
}
class Cube(Float width) satisfies Figure3D {
    shared actual Float area = width * width;
    shared actual Float depth = width;
}
class Cylinder(Integer radius, Float depth)
        satisfies Figure3D {
    shared actual Float area = 3.14 * radius ** 2;
    shared actual Float depth = depth;
}
```

OMG multiple inheritance mess!?

- No state (initialization)
  - No ordering issues
  - A single superclass
- Must redefine a method if ambiguous



Ceylon is extremely regular

```
Integer attribute = 1;
Integer attribute2 { return 2; }
void method(){}
interface Interface{}

class Class(Integer x){
    Integer attribute = x;
    Integer attribute2 { return x; }
    class InnerClass(){}
    interface InnerInterface{}

    void method(Integer y){
        Integer attribute;
        Integer attribute2 { return y; }
        class LocalClass(){}
        interface LocalInterface{}
        void innerMethod(){}
    }
}
```

Hierarchical structure

```
Table table = Table {
    title = "Squares";
    rows = 5;
    border = Border {
        padding = 2;
        weight = 1;
    };
    Column {
        heading = "x";
        width = 10;
        String content(Integer row) {
            return row.string;
        }
    },
    Column {
        heading = "x**2";
        width = 12;
        String content(Integer row) {
            return (row**2).string;
        }
    }
};
```

Formal mathematical proof of the type and effect system

Semantics 1/154

$$
\begin{array}{l}
\mathcal{E}[\![(\texttt{lambda}\ \ (\mathrm{I}^*\ .\ \mathrm{I})\ \ \Gamma^*\ \ \mathrm{E}_0)]\!] = \\
\quad \lambda\rho\kappa\,.\,\lambda\sigma\,. \\
\qquad new\,\sigma \in \mathtt{L} \rightarrow \\
\qquad\quad send\,(\langle new\,\sigma \,|\, \mathtt{L}, \\
\qquad\qquad \lambda\epsilon^*\kappa'\,.\,\#\epsilon^* \geq \#\mathrm{I}^* \rightarrow \\
\qquad\qquad\quad tievalsrest \\
\qquad\qquad\qquad (\lambda\alpha^*\,.\,(\lambda\rho'\,.\,\mathcal{C}[\![\Gamma^*]\!]\rho'(\mathcal{E}[\![\mathrm{E}_0]\!]\rho'\kappa')) \\
\qquad\qquad\qquad\qquad (extends\,\rho\,(\mathrm{I}^*\,\S\,\langle \mathrm{I}\rangle)\,\alpha^*)) \\
\qquad\qquad\qquad \epsilon^* \\
\qquad\qquad\qquad (\#\mathrm{I}^*), \\
\qquad\qquad\quad wrong\ \text{“too few arguments”}\rangle\ \text{in}\ \mathtt{E}) \\
\qquad \kappa \\
\qquad (update\,(new\,\sigma \,|\, \mathtt{L})\,unspecified\,\sigma), \\
wrong\ \text{“out of memory”}\ \sigma
\end{array}
$$

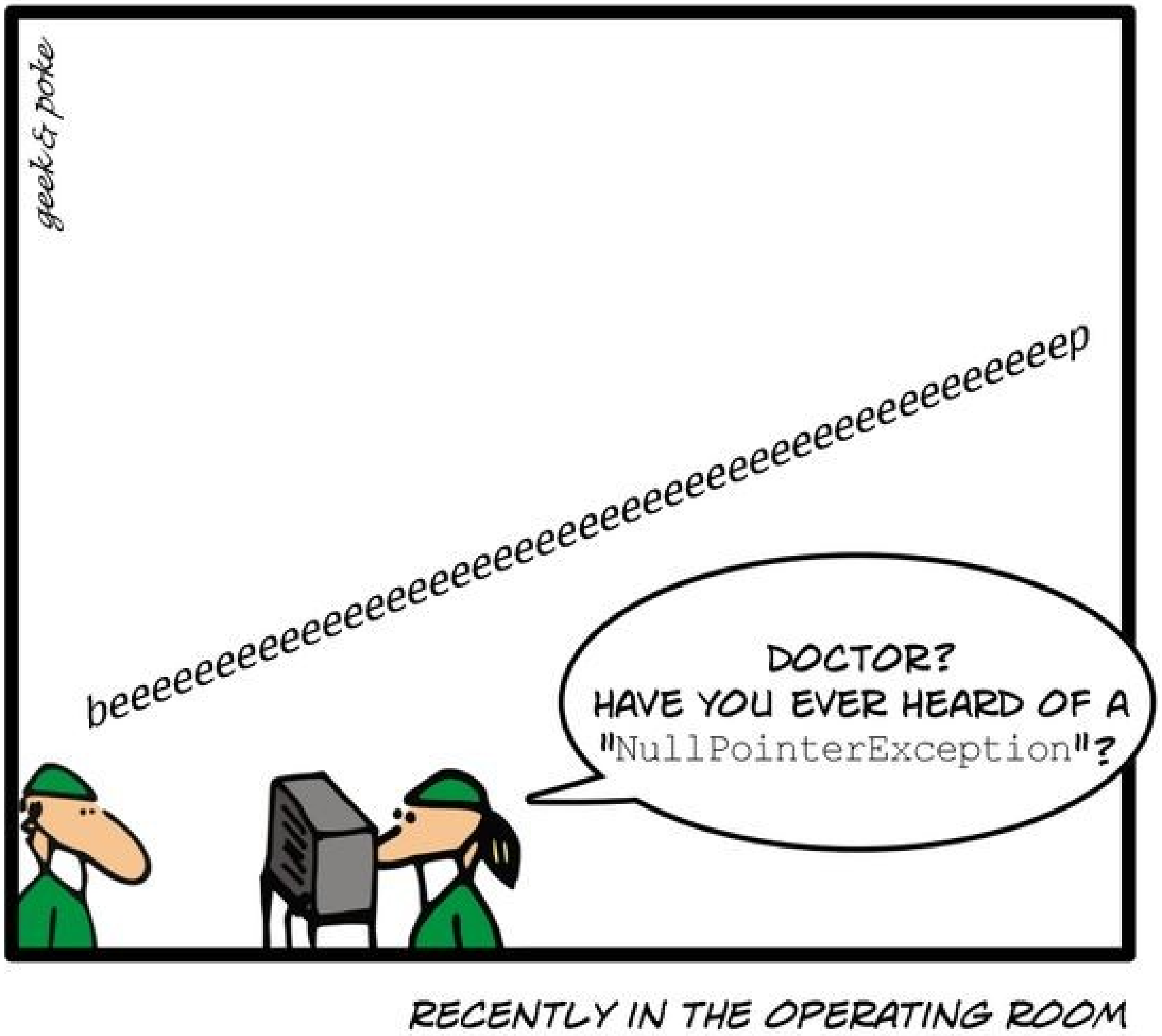
Just Kidding!

Typical types

```
Integer n = 10.times(2);       // no primitive types
String[] s = {"foo", "bar"}; // inference
Number[] r = 1..2;             // intervals
```

```
// inference
function makeCube(Float width){
    return Cube(width);
}
value cube2 = makeCube(3.0);
```

Death to NPEs



Type safely

```
// optional?
Cube? cubeOrNoCube() { return null; }
Cube? cube = cubeOrNoCube();

print(cube.area.string); // compile error

if(exists cube){
    print(cube.area.string);
}else{
    print("Got no cube");
}
```

Some sugar on top?

```
// default value
Cube cube2 = cubeOrNoCube() else Cube(2.0);
// nullsafe access
Float? area = cube?.area;
```

Operations on lists

```
Integer[] numbers = {1,2,3};
// slices
Integer[] subList = numbers[1..2];
Integer[] rest = numbers[1...];
// map/spread
Integer[] successors = numbers[].successor;
Integer[] shifted = numbers[].minus(2);
```

Functional programming

```
// using closures (FP-style)
value urls = map.keys
    .filter(function(String key) key.contains("url"))
    .map(function(String  key) map.item(key));

// using comprehensions (Imperative-style)
value urls2 = { for(key in keys)
                  if(key.contains("url"))
                   json.get(key) };
```

(some of) Typing

Union type

- To be able to hold values among a list of types
- We must check the actual type before use
  - `TypeA|TypeB`
  - `Type?` is an alias for `Type|Nothing`

Union type example

```
class Apple() {
    shared void eat(){}
}
class Garbage() {
    shared void throwAway(){}
}
void unions(){
    Sequence<Apple|Garbage> boxes = {Apple(), Garbage()};
    for(Apple|Garbage box in boxes){
        print(box.string);
        if(is Apple box){
            box.eat();
        }else if(is Garbage box){
            box.throwAway();
        }
    }
}
```

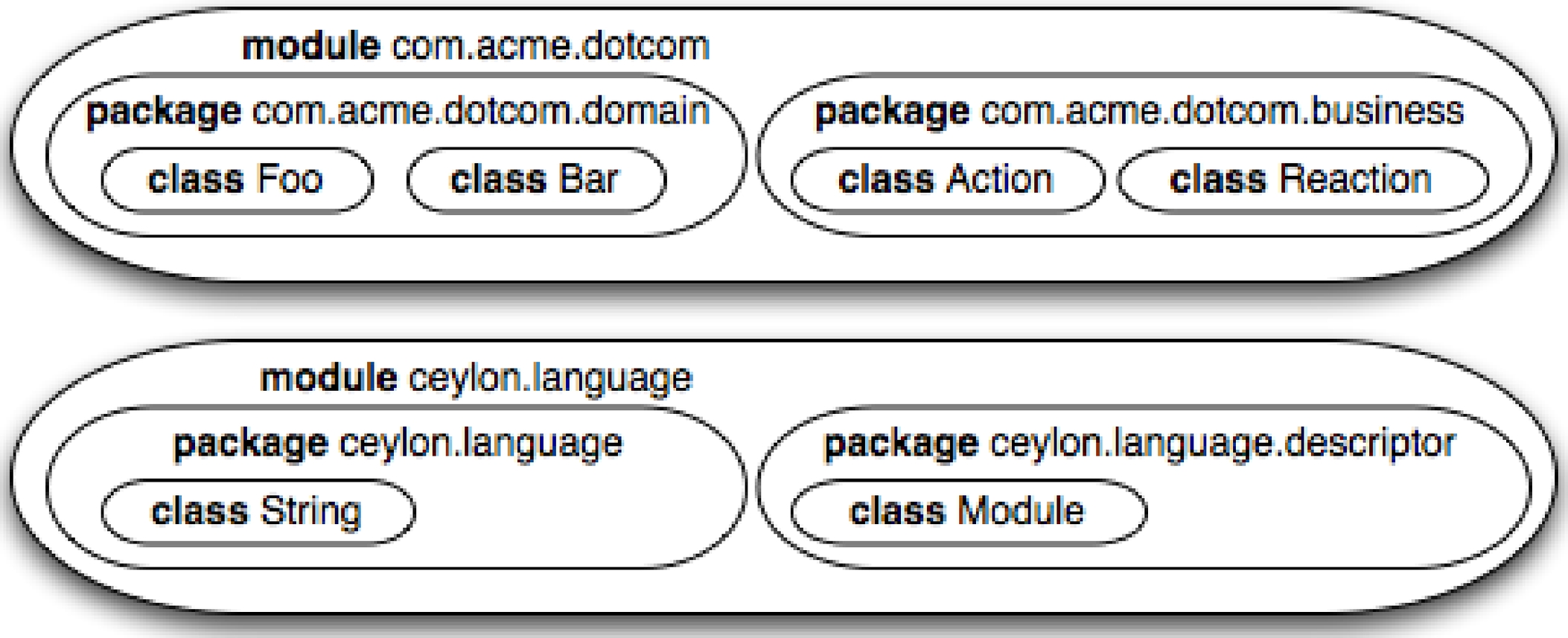
Intersection type

```
interface Food {
    shared formal void eat();
}
interface Drink {
    shared formal void drink();
}
class Guinness() satisfies Food & Drink {
    shared actual void drink() {}
    shared actual void eat() {}
}
void intersections(){
    Food & Drink specialStuff = Guinness();
    specialStuff.drink();
    specialStuff.eat();
}
```

A lot more features

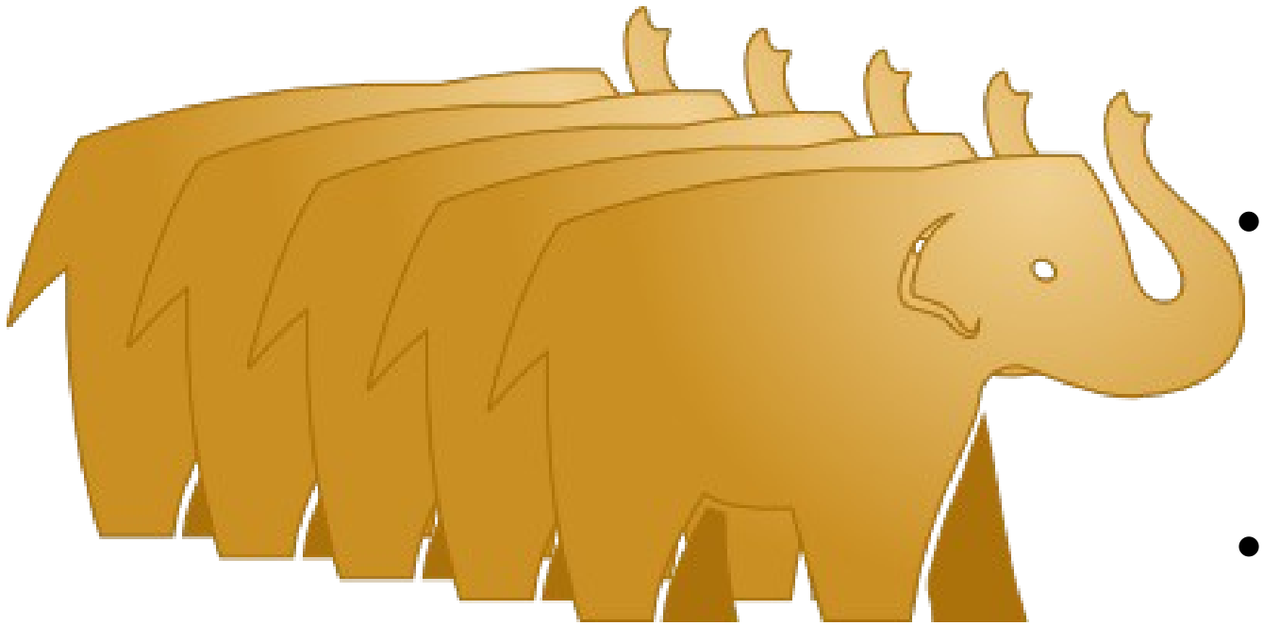
- Type parameters
- Singletons and anonymous classes
- Introductions
- Attribute and method references
- Partial application
- Annotations
- Type aliases
- Meta-model
- Interception

Modularity



- Core to the language
- Integrated in the tool chain

Herd



- Our next-gen module repo
- On http://modules.ceylon-lang.org
  - Already available, and usable from the tools
- Intuitive and good-looking interface *à-la* Github
  - Collaborative
- Free Software
  - Private repos encouraged

Demo !

With some IDE inside

* May contain traces of Herd

Community

- Completely open
- Some from JBoss/RedHat
- And (very) active contributors
  - From all over
- And you!

A fun project

- Nice people :)
- Best tools
  - github, ant, Eclipse, HTML5, Awestruct, Java, JavaScript, OpenShift, Play!
- Many subprojects
  - spec, typechecker, Java compiler, JavaScript compiler, Eclipse IDE, Web IDE, Herd, module system, ceylondoc, Ant/Maven plugins

To infinity…

- Five milestones to reach 1.0
- Some features targeted to 1.1

- M1 (done)
  - Minimum all Java-style features
  - All the tools (with the IDE)
- M2 (done)
  - Interoperability with Java
  - Enumerated types
  - First-class methods

…and beyond!

- M3 (done)
  - **TODAY :)**
  - Anonymous functions
  - IO, math modules
  - Mixin inheritance
  - Comprehensions
- M4
  - Member classes refinement
  - Type families
  - Type aliases
- M5 (Ceylon 1.0)
  - Annotations
  - Reified generics
  - Metamodel
  - Interception

How to find us

- Our website http://ceylon-lang.org
  - Blog, introduction, tour, reference, spec, API, downloads
  - Herd: http://modules.ceylon-lang.org
- Source repositories
  - http://github.com/ceylon
- Development/user mailing list
  - Google groups: ceylon-dev, ceylon-users
- Google+: http://ceylon-lang.org/+
- Twitter: @ceylonlang

Q&A

- Questions! Answers?
- http://ceylon-lang.org